# Uvod

Cesto prilikom ovih system design intervjua ume da se desi I da nema tacnog odgovora nego se testira kompunikacija I problem solving skills pa vole da ih ubacuju u intervjue.

Bices ocenjen na osnovu toga koliko dobro analiziras nejasne probleme I kako resaves te probleme korak po korak.   
Takodje testira se i kako objasnjavas ideje, razgovaras sa drugima, evaluiras i optimizujes sistem.

The system design questions are open-ended.

Smer intervjua zavisi od osobe koja te intervjuise. Neki zele high level arhitekturu da se pokriju svi apsekti dok neki zele fokus na odredjenu sekciju.

Tipicno, system requirements, constraints & bottlenecks sistema koji “osmisljavas” treba dobro da se razumeju kako bi se oblikovao smer intervjuera i intervjuisanog.

## Chapter 1: Scale from zero to millions of users

Ovde cemo napraviti sistem koji podrzava jednog usera i lagano ga scaleovati na milione usera.

### Single server setup

Ovo je prvi korak.

Da bi zapoceli sa necim jednostavnim, prvomoramo imati jelte jednostavno.   
Slika ispod predstavlja single server setup gde je sve na jednom server; web app, db, cache i slicno.

A diagram of a computer system

AI-generated content may be incorrect.

Sta se desava:

1. Prvo user posalje DNS(Domain Name Server)-u upit za dobijanje IP adrese naseg servera. DNS je paid service kog daje neki 3rd party i ne hostuje se nasim serverima.
2. IP(Internet Protocol) adresa se vraca uredjaju.
3. Jednom kad je IP adresa pridobijena, salje se HTTP(Hyper Text Transfer Protocol) request direktno na tvoj server
4. Web server vraca HTML stranicu/JSON response za renderovanje

Pogledajmo traffic source. To su web application & mobile application.

* Web Application koristi kombinaciju server side jezika da resava “biznis logiku”, skladistenje i client side jezika(HTML & JS) za prezentaciju.
* Mobile application: HTTP protocol je komunikacioni protokol izmedju mobilne aplikacije i web servera. JSON je cesto koristeni API response format.

### Database

Recimo rastu ti korisnici. Jedan server nije dovoljan i sad nam treba vise servera. Jedan za web/mobile traffic, a drugi za bazu. Razdvajanjem web/mobile traffic(web tier) & baze(data tier), serveri sada mogu nezavisno da skaliraju.

A diagram of a computer network

AI-generated content may be incorrect.

### Koju bazu koristiti

Relaciona & nerelaciona.

Relacione znas vec sta su.   
Nerelacione(NoSQL)(CouchDB, Neo4j, Cassandra, HBase, Amazon DynamoDB) se dele u x4 kategorije:

1. Key-value stores
2. Graph stores
3. Column stores
4. Document Stores

Join operacije ovde cesto nisu podrzane.

Razlozi da se odlucis za nerelacione baze:  
- Treba ti super low latency  
- Podaci su ti nestruktuisani ili nemas “relacione” podatke  
- Trebas samo da serializujes i deserializujes podatke(JSON/XAML/YAML etc..)  
- Trebas da skladistis velike(massive) kolicine podataka

### Vertical vs Horizontal scale

Vertical scaling(scale up) znaci dodavanje snage procesorom/ramo memorijom na serveru.

Horizontalno(scale out) znaci dodavanje servera u tvoj pool resursa.

Kad je saobracaj nizak, vertikalno skaliranje je skroz okej i jednostavnost ove opcije je njena glavna prednost. Nazalost, ima ogranicenja:  
- Ima hard limit. Nemoguce je dodati “bezbrojno” CPU-a i RAM-a na single server  
- Vertical scale nema failover i redundanost. Ako jedan server pukne, puca sve.

Horizontalno skaliranje je pozeljnije za vece aplikacije upravo zbog ovih ogranicenja.

Znaci u gore primeru su se korisnici kacili na x1 server. Ako je taj server nedostupan, ne mozes mu pristupiti. Ako se vise korisnika prikaci na server istovremeno i dostignes web serverov load limit, korisnici iskuse generalno sporiji response ili ne mogu da se povezu na server.

Za taj problem, resenje je **LOAD BALANCER.**

### Load balancer

Load balancer podjednako rasporedjuje dolazeci saobracaj izmedju web servera koji su definisani u “load-balanced set”.

A diagram of a network

AI-generated content may be incorrect.

U gore slici je definisano kako load balancer izgleda.

Korisnici se povezuju na public IP load balancera direktno. Na taj nacin, web serverima se vise ne pristupa direktno od strane klijenata. Za bolji security, private IP’s se koriste za komunikaciju izmedju servera.

Private IP je IP adresa kojoj mogu pristupati samo serveri unutar iste mreze, ali ne i van te mreze(e.g. internet).

Posle load balancera i dodavanja drugog servera, sada smo resili problem failovera i poboljsali dostupnost **web tier**-a.

* Ako server 1 ode offline, sav saobracaj usmerice se na server 2. Na taj nacin website nece biti offline. Takodje dodacemo novi zdravi server u server pool da bi se balansiralo opterecenje
* Ako website saobracaj naglo poraste i dva servera nisu dovoljna da izdrze taj saobracaj, load balancer moze ovaj problem lako da sredi dodavanjem novih servera u web server pool kojim ce LB kasnije balansirati.

To je nas **web tier**. Sta je sa **data tier**? Trenutni dizajn ima samo x1 database bez failovera i redundantnosti. Database replication je cesta tehnika za resavanje tih problema.

### Database replication

Database replication se cesto koristi u master/slave odnosu izmedju originala(master) i kopija(slaves) baza.

Master baza cesto podrzava samo write/update/delete operacije. Slave baza dobija kopije podataka od master baze i podrzava samo read operacije. Sve data modifying komande poput insert, delete ili update moraju biti poslate na master bazu.

Vecina aplikacija zahteva mnogo veci odnos u reads naspram writes pa je cesto i broj slave baza u sistemu bezi od broja master baza.

Na slici ispod moze se videti master baza naspram slave baza.

A diagram of a computer server

AI-generated content may be incorrect.

Prednosti database replicationa:

* **Bolje performanse**. read operacije se distribuiraju across slave nodes jer sada mozes vise upita procesirati paralelno.
* **Reliability**(pouzdanost). Ako jedan db server padne, ne moras brinuti o data loss-u jer su podaci replicirani duz sistema.
* **Visoka dostupnost**. Repliciranjem podataka kroz vise ralzicitih lokacija, website ti moze biti dostupan cak iako jedna baza padne.

U predhonom primeri imali smo load balanser koji je pomagao da se preusmeri saobracaj. Kako to ovde?

* Ako je samo jedna slave baza dostupna i ona ode offline, read operacije ce se privremeno preusmeriti ka master bazi. Cim se problem pronadje, nova slave baza ce zameniti staru. U slucaju da imamo vise slave baza, one ce uzeti taj saobracaj.
* Ako master baza ode offline, slave baza ce biti promovisana kao nova master baza. Sve db opecaije se privremeno biti izvrsene na novoj master bazi. Nova slave baza ce zamniti staru za data replication. U production systems, promotion je kompleksniji jer novi master mozda nema sve podatke pa je neophodno uz recovery scripts podici bazu da bude up to date. Postoje multi-master & circular replication metode koje mogu pomoci, ali to je out of scope za ovu knjigu.

A diagram of a computer network

AI-generated content may be incorrect.

Arhitektura nakon dodavanja replikacije u nas sistem.

Ukratno, jasno ti je sta se ovde desava. Sad kad se malo bolje razumemo u web/data tiers, mozemo da predjeno na sledecu tacku, a to je poboljsanje load & response times. Dodavanjem cache layera i shifting static contenta(js/css/video files) u CDN(Content Delivery Network) ce nam tu mnogo pomoci.

### Cache

Cache je privremeni storage area koja cuva rezultate “skupih” upita ili cesto pristupanim podacima u memoriji tako da se requestovi brze opsluzuju. U gore slici, svaki put kad se nova web stranica ucitava, neophodno je odraditi jedan ili vise db poziva. Performanse aplikacije su itekako zahvacene ovim.

### Cache tier

Cache tier je privremeni data store layer mnogo brzi od baze.   
Benefiti:  
- rasterecenija baza  
- bolje performanse sistema  
- mogucnost skaliranja cache tiera nezavisno

A diagram of a computer

AI-generated content may be incorrect.

Nakon dobijanja requesta, web server prvo proverava ako cache ima dostupan resposne. Ako ima, onda se vrati. Ako nema, onda se query upucuje bazi, rezultat se memorise u cache i uzvraca se serveru.

Ta napomenuta **caching strategy** se naziva **read-through cache.** Druge strategije su dostupne u zavisnosti od tipa podataka, velicine i obrasca pristupanja(access patterns).

Vecina cache servera nude API za vecinu programskih jezika pa je i interakcija sa istima poprilicno jednostavna.

Primer za **memcached:**  
A close up of a text

AI-generated content may be incorrect.

### Considerations for using cache

* **Odluci o tome kada zelis da koristis cache**.  
  Valja za scenario u kom imas puno citanja, a malo modifikacija. Cache nije idealan za cuvanje podataka jer je “volatile memory” u pitanju. Ako se cache server restartuje, gubis sve podatke.
* **Expiration policy.** Dobra je praksa imati expiration policy kojom jednom kada cachirani podaci isteknu, ciste se iz cachea
* **Consistency.** Ovo ukljucuje sinhronizaciju data stores-a i cachea. Inconsistency se desava kada data modifying oepracije na data store & cache nisu u single transaction. Skaliranjem preko “vise regiona” je izazovno. Facebook ima “Scaling memcache at facebook” document.
* **Mitigating failures.** Cache server i dalje moze da pukne i da bude potencijalni single point of failure(SPOF). SPOF znaci da je to u pitanju deo sistema koji kada pukne, moze da znaci da ceo sistem puca. Visestruki cache serveri u razlicitim data centrima su recommended. Drugi recommendation je da pogledas koji ti je required memory i uvek imas buffer iznad za odredjeni %. A diagram of a router

  AI-generated content may be incorrect.
* **Eviction policy.** Jednom kad je cache full, zahtev da se dodaju itemi u cache mogu rezultovati da se postojeci itemi obrisu. To se naziva **cache eviction**. **Least recently used(LRU) je najcesci eviction policy**. Postoje **i Least Frequently Used(LFU)** ili **First in First Out(FIFO).**

### CDN - Content Delivery Network

CDN je mreza globalno distribuiranih servera koji se koriste da dostave staticki sadrzaj. CDN serveri uglavnom taj sadrzaj cacheuju(slike, video, js, css, html pages etc.)

**Dynamic content caching** je relativno nov koncept i van opsega knjige. Uglavnom ideja je da se na osnovu request path, query strings, cooking i request headera omogucava caching HTML stranica.

Ovo je kako CDN funkcionise na high level. Kad korisnik pristupa websiteu, cdn server najblizi korisniku ce dostaviti staticki sadrzaj. Intitivno, sto su dalji korisnici od CDN-a, to se sajt sporije ucitava. NPR CDN serveri koji su u San Francisko, a korisnici u LA ce dobiti sadrzaj brze od korisnika u EU.

A diagram of a computer network

AI-generated content may be incorrect.Slika 1-9 koja pokazuje cdn na high level.

A diagram of a cloud computing system

AI-generated content may be incorrect.

Slika 1-10 CDN workflow.

U sustini slicno kao cache. Korisnik posalje zahtev za neku sliku, cdn prihvati zahtev i ukoliko nema sliku u cache-u, onda se prosledjuje zahtev serveru. Rezultat servera(slika) se skladisti u CDN i vraca korisniku. Svaki sledeci korisnik koji posalje zahtev za tu sliku ce je dobiti direktno od CDN-a bez da komunicira sa serverom.

Da se razumemo, taj “server”, moze da bude i online storage poput amazon S3. U pitanju je origin jelte. U trenutku kada “server” vraca sliku CDN-u, on moze opciono postaviti TTL HTTP header koji naznacava koliko dugo ce se slika cacheovati.

### Considerations of using a CDN

**Cost**: CDN’s su u vlasnistvu 3rd party-a i placas data transfers “in and out” of cdn.  
Caching infrequently used assets ti ne daje neki benefit ako koristis CDN.

**Setting an appropriate cache expiry**: Za time sensitive content cache expiry je bitan i treba da bude ne previse dugacak(outdated) niti previse kratak(repeat reload)

**CDN Fallback**: Testiraj kako ti se app ponasa ako CDN padne. Ukoliko se to desi, clienti bi trebalo da mogu da detectuju problem i da se zahtevi preusmere direktno u origin za dobijanje resursa.

**Invalidating files:** Mozes ocistiti file iz CDN pre nego istekne sledecim operacijama:  
- Invalidate using CDN provider API  
- Use object versioning to serve a different version of the object. Za verzionisanje mozes dodati version query parameter poput **image.png?v=2**

A diagram of a computer network

AI-generated content may be incorrect.

Ovo je arhitektura nakon sto dodamo CDN & Cache.

1. Static assets se sada ucitavaju od strane CDN-a
2. DB load je rasterecen koriscenjem cache data

### Statless web tier

Sada cemo da skaliramo web-tier horizontalno. Da bi to uradili, prvo moramo pomeriti “state” van web tiera. Pod “state” mislimo na podatke poput session data. Dobra praksa je cuvanje session data unutar persistent storage poput relacionih baza ili nosql.

Svaki web server u clusteru moze pristupiti tim podacima iz baza. To se naziva stateless web tier.

### Stateful architecture

Stateful server & stateless server imaju razlike.

Stateful “pamti” dok stateless ne pamti client podatke izmedju requestova(from one request to the next).

A diagram of a computer server

AI-generated content may be incorrect.

Primer stateful arhitekture.

Server 1 ce npr pamtiti session data od usera A itd. Da bi se autentikovao user A, http client mora biti preusmeren na server 1 jer ako je usmeren u server 2, onda ce failovati jer server 2 nema info o user A.

To se moze resiti uz “sticky sessions” u vecini load balancera, ali to dodaje neki overhead i teze se handleuje failure pa je lakse jednostavno sve persistovati u data tier/storage umesto na serveru.

### Stateless architecture

A diagram of a computer server

AI-generated content may be incorrect.

U statelessu, zahtev moze da se salje na bilo koji server jer se podaci o sesijama cuvaju na shared storage van web servera.

A diagram of a computer server

AI-generated content may be incorrect.

Ovom arhitekturom smo otklonili state iz servera i dodali smo persistent data store. U ovom slucaju **NoSQL** data store je odabran za tako nesto.

Autoscaling znaci dodavanje ili otklanjanje web servera automatski na osnovu opterecenja u saobracaju.

**Sada imas znacajan broj korisnika globalno. Da bi poboljsao dostupnost i user experience duz sire geografske zone, neophodno je imati vise data centara.**

### Data centers

A diagram of a computer network

AI-generated content may be incorrect.

Ovde je prikazan primer arhitekture sa vise data centara. Jedan DC1 u US East & DC2 u US West. Korisnici su geoDNS-routed najblizem centru sa traffic split od **x%**

**geoDNS** je DNS servis koji dozvoljava da se domain names resolveuju na IP adrese na osnovu lokacije korisnika.

U slucaju da data centar postane nedostupan, sav saobracaj ce se usmeriti ka zdravo data centru.

Kako bi uspostavili multi-data center setup, postoji nekoliko tehnickih izazova koji moraju da se razrese.

**Traffic redirection:** Neophodni su efektivni alati kako bi se usmerio saobracaj ka ispravnom data centru. GeoDNS je npr jedan od njih.

**Data synchronization:** Korisnici iz razlicitih regiona mogu koristiti razlicite lokalne baze ili cache. U failover cases, saobracaj bi se routeovao u data centar gde su ti podaci nedostupni. Cesta strategija je replikacija duz vise data centara.

**Test and deployment:** Sa multi-data center postavkom, bitno je testirati website/application na razlicitim lokacijama. Koristi automated deployment tools za postizanje consistency-a.

Za dalje skaliranje, neophodno je razdovjiti sledece komponente.

Messaging queue je kljucna strategija koju koriste skoro svi distribuirani sistemi.

### Message queue

Message queue je durable component stored in memory koja podrzava asinhronu komunikaciju. Sluzi kao buffer i distribuira asinhrone zahteve.

Osnovna arhitektura message queue-a je takva da imas input services(producers/publishers) koji kreiraju poruke, publishuju ih u queue i potom drugi servisi(consumers/subscribers) se povezuju na queue, citaju poruke i vrse “nesto” na osnovu tih poruka.

A blue and white line of email icons

AI-generated content may be incorrect.

Producer moze da publishuje poruke i kad consumer nije dostupan.  
Consumer moze da slusa poruke i kada producer nije dostupan.

Scenario:  
Aplikacija koja nudi photo customization. Serveri pushuju customization taskove u message queue koje kasnije photo processing workeri konzumiraju. Producer & consumer se mogu nezavisno skalirati. Ukoliko ima previse poruka, workeri se skaliraju. Ukoliko nema poruka, smanjuje se broj workera.

### Logging, metrics, automation

Logging, metrics & automation support su generalno dobre prakse, ali kada radimo sa malim sajtom koji radi na par servara i nisu toliko nephodne.

Sad zamisli da imas veliki “biznis”. Onda postaju krucijalni.

**Logging**: Monitoring error logova zarad identifikovanja gresaka i problema u sistemu.

**Metrics:** Skupljanje razlicitih metrika nam dozvoljava da dobijemo povratne informacije o poslovanju i razumemo “health status” sistema. **Host level metrics**(cpu health, mem usage), **Key business metrics**(daily active users, retention, revenue) & **Aggregated level maetrics**(npr performanse celog data tiera/cache tiera i slicno)

**Automation:** Kada sistem postane velik i kompleksan, neophodna nam je automacija da bi poboljsali produktivnost. CI je dobra praksa u kojoj je svaki code check-in verifikovan i problemi se ranije detektuju. Automatizacija build, test & deploy procesa itekako povecava produktivnost.

### Adding messages to queues & different tools

Slika ispod predstavlja updateovani dizajn(bice prikazan samo x1 data center jer jbg prostor)

1. Ukljucen je message queue kako bi se sistem bolje razdvojio i postao otporniji na failures
2. Logging, monitoring & automation tools su ukljuceni

A diagram of a computer network

AI-generated content may be incorrect.

I sada kako podaci rastu, baza je pod vise opterecenja - vreme je da se skalira data tier.

### Database scaling

Isto kao i sa web tier, mozemo horizontalno i vertikalno skalirati baze.

### Vertical scaling

Ya know. Amazonov RDS(Relational database server) nudi servere sa 24 TB RAM memorije sto je jako kul. Stackoverflow na primer uspeo je da ima 10 miliona mesecnih unique posetioca i to je sve sa x1 master bazon.

Mane su standardne, povecana cena, smanjena redundantnost(ako ti padne, pada sve) i limiti hardvera. Single instance can go up only so far.

### Horizontal scaling(Sharding)

Takodje poznato kao sharding je kada dodajes vise servera horizontalno. Shardingom se velike baze razdvajaju u manje, jednostavnije delove koji se nazivaju “shards”. Svaki shard deli istu schemu, ali podaci izmedju shardova se razlikuju. Podaci na svakom shardu su unique za shard.

Na slici dole mozemo videti primer shardovne baze. User data se alocira u db server na osnovu user ID-a. Svaki put kad pristupas podatku, hash funkcija se koristi da se pronadje odgovarajuci shard. U nasem primeru, user\_id % 4 se koristi kao hash funkcija.

Ako je rezultat 0, koristis shard 0 da storeujes i fetchujes podatke za tog usera. Itd.

A diagram of a diagram of a diagram

AI-generated content may be incorrect.

A screenshot of a table

AI-generated content may be incorrect.

Najbitniji faktor koji treba razmotriti prilikom implementiranja sharding strategije jeste odabir sharding key-a.

**Sharding key(partition key)** se sastoji iz jedne ili vise kolona koje odlucuju o tome kako se podaci distribuiraju. Na primer na slici gore, “user\_id” je sharding key.

Sharding key ti dozvoljava da pribavljas i modifikujes podatke efikasno rutiranjem db querya ka ispravnoj bazi.

Pri biranju sharding key-a, jedan od najbitnijih kriterijuma koje treba da odaberes jeste da odaberes kljuc koji moze podjednako distribuirati podatke. Sharding je dobra tehnika za skaliranje baza, ali daleko je od savrsene.

Postoje kompleksnosti poput:

* **Resharding data**: neophodno je kada single shard ne moze vise podataka da drzi u sebi zbog naglog rasta ili kada odredjeni shardovi mogu iskusiti “shard exhaustion” brze od drugih zbog neravnomerne distribucije podataka. Kada se ovo dogodi, razmotri u update sharding funkcije i pomeranju podataka. **Consistent hashing** je tehnika koja se cesto koristi za resavanje ovog problema.
* **Celebrity problem:** Takodje poznat kao **hotspot key problem**. Prekomeran pristup odredjenom shardu moze prouzrokovati server overload. Zamisli da su justin bieber, katy perry & lady gaga na istom shardu, a ti imas neki social app. Resavanje ovog problema znaci dodavanja posebnog sharda toj poznatoj osobi, ali svaki shard mozda bude zahtevao i dodatno particionisanje.
* **Join & de-normalization**: Jednom kad se baza sharduje across multiple server, teze je izvrsiti join operacije izmedju vise razlicitih db shardova. Cest workaround jeste denormalizacija baza kako bi se queryji mogli izvrsiti unutar jedne tabele.

Na sledecoj slici shardujemo baze da bi podrzali rapidly increasing data traffic. U isto vreme, neke od nerelacionih funkcionalnosti su pomerene u NoSQL data store kako bi smanjili DB load.

A diagram of a computer server

AI-generated content may be incorrect.

### Millions of users & beyond

Skaliranje sistema je iterativan proces. Iteracija na osnovu naucenog nas moze dogurati do “odredjene tacke”.

Vise “fine tuning” i noivih strategija su neophodne da bi skalirali preko milion korisnika, npr mozda je neophodno optimizovati sistem i razdvojiti sistem u jos manje servise. Sve naucene tehnike bi trebale omoguciti dobru osnovu.

Summary chaptera:

* Drzi web tier stateless
* Gradi redundantnost na svakom tieru
* Cacheuj podatke sto vise mozes
* Podrzi vise data centara
* Hostuj staticke podatne na CDN-u
* Skaliraj data tier shardovanjem
* Razdvajaj tierove u njihove individualne servise
* Vrsi monitoring nad sistemom i koristi alate za automaciju

A screenshot of a computer

AI-generated content may be incorrect.

## Chapter 2: Back-of-the-enevlope estimation

U sys design untervjuima ces nekad biti pitan da estimiras system capacity ili performance requirements koriscenjem back of the envelope estimacije(estimacija poledjine koverte)

“back of the envelope” kalkulacije su estimacije koje kreiras koriscenjem kombinacije misaonih eksperimenata i “common” performance numbers(cesti performans brojevi) da bi dobio osecaj za dizajn koji bi odgovarao tvojim zahtevima.

Neophodno je prvo steci dobar smisao za skalabilnost. Naredni koncepti ce ti pomoci:

* power of two
* latency numbers every programmer should know
* availability numbers

### Power of two

Iako volumen podataka moze biti ogroman kada se radi o distribuiranim sistemima, kalkulacije se zasnivaju na osnovama.

Da bi ih dobro proracunao, bitno je da znas “data volume unit using the power of 2”.

1B je 8b. ASCII karakter koristi 1B memorije.

1KB => 2^10  
1MB => 2^20  
1GB => 2^30  
1TB => 2^40  
1PB => 2^50

### Latency numbers every programmer should know

Neki brojevi su outdated, ali otprilike bi trebalo da ti daju smisao

A screenshot of a computer

AI-generated content may be incorrect.

Brojevi za vizuelizaciju ovih gore metrika:

A screenshot of a computer

AI-generated content may be incorrect.

Neki summary:

* Memorija je brza, ali disk je spor
* Izbegavaj disk seek ako je to moguce
* Jednostavni compression algoritmi su brzi
* Kompresuj podatke pre nego ih saljes preko mreze ako je to moguce
* Data centri su cesto u razlicitim regionima i treba vremena za razmenu podataka izmedju njih

### Availability numbers

Visoka dostupnost je sposobnost sistema da se kontinualno izvrsava sto duze. Visoka dostupnost se izrazava u procentima gde 100% znaci 0 downtime.

Vecina servisa su izmedju 99% i 100%.

SLA(service level agreement) je cesto koristeni izraz za servis providere koji predstavlja “agreement”(dogovor) izmedju tebe i service providera(jelte) koji formalizuje nivo uptime-a tvog servisa.

Amazon, google & msft nude SLAs na 99.9% i iznad. Uptime je tradicionalno meren u “devetkama”(9). Sto vise devetki, to bolje.

A screenshot of a white screen

AI-generated content may be incorrect.

### Example: Estiamte twitter QPS & storage requirements

Assumptions:

* 300 million active monthly users
* 50% of users use twitter daily
* users post 2 tweets/day average
* 10% of tweets contain media
* data is stored for 5 yrs

Estimation:

Query per second(QPS) estimate:

* daily active users = 300 million \*50% =150 million
* Tweets QPS = 150 million \*2 /24h /3600 = ~3500
* Peek QPS = 2\*QPS = ~7000

We only estimate media storage here:

* avg tweet size
  + tweet\_id 64B
  + text 140B
  + media 1MB
* Media storage: 150 million \*2 \*10% \*1MB = 30TB/day
* 5 year media storage => 30TB \* 365 \* 5 = ~55PB

Back of the envelope estimation je all about the process.  
Resavanje problema je bitnije nego da dobijes skroz tacne rezultate. Invervjueri mozda testiraju tvoj problem-solving skills.

Evo par saveta:

* **Zaokrugljivanje & estimacije.** Nema potrebe da se sve skroz mega precizno izracuna. Recimo kalkulacija poput 99987/9.1. Jeste sigurno. Zaokrugli. “100,000/10”
* **Zapisi svoje pretpostavke**, za reference kasnije
* **Oznaci svoje jedinice(units)**. Kad napises 5, reci sta je to. Dan, KB, godina?
* **Cesto pitana back of the envelope estimation pitanja**:
  + **QPS**
  + **peak QPS**
  + **storage**
  + **cache**
  + **number of servers etc.**

Mozes vezbati ove kalkulacije i ovakve zadatke. Koristi chatgpt.

## Chapter 3: framework za system design interview

Dobra stvar kod ovih sys design intervjua je sto ne ocekuju da sad odgovoris tacno na pitanje tj da dizajniras popularni product.

Real world system design je ekstremno komplikovan. Npr google serach je naizgled jednostavna, ali kolicina tehnologije koja se desava u pozadini je jeziva.

System design intervju sluzi da testira tvoj real life problem solving gde dvoje kolega saradjuju na nekom problemu i dodju do resenja koje ispunjava njihove zahteve.

Krajnji dizajn je manje bitan u poredjenju sa “work you put in during the design process”.

Bitno je znati psotaviti dobra pitanja i vecina intervjuera traze tu vestinu.

Takodje traze se i red flags. Over engineering je cesta “bolest” vecine inzenjera jer ulaze u “design purity” i ignorisu tradeoffs. Tu postoji i cena u over-engineeringu koju kompanija placa zbog tudjeg neznanja. To ne zelis.

Evo nekih saveta za intervjue:

### 4 Step process for effective system design interview

Postoje neki koraci koje svi sistem dizajn intervjui mogu da dele, evo ih.

#### Step 1: Understand the problem and establish design scope

Jimmy je neko ko instant odgovori na svako pitanje, misli da zna sve i na pitanje “zasto je tigar jauknuo”, on odgovara sa “bio je gladan” na sta mu profesor kaze “tacno”.

**Ne budi “Jimmy”.**

Ne dobijas bonus poene ako odmah das odgovore na pitanja. Odgovaranje bez razumevanja zahteva je **huge red flag.**

Zato nemoj odmah davati odgovor na pitanje, uspori i postavi pitanja da bi bolje razumeo zahteve i pretpostavke. To je najvazniji deo.

**Kada postavljas pitanja**, intervjuer ce cesto odgovoriti direktno na tvoje pitanje ili te pitati da napravis svoju pretpostavku. Ako se ovo drugo desi, napisi svoje pretpostavke na papir ili tablu, mozda ti zatrebaju.

**Koja pitanja postavljati? Ona koja ce ti pomoci da razumes tacne zahteve:**

* “What specific features are we going to build”
* “How many users does the product have”
* “How fast does the company anticipate to scale up? What are the anticipated scales in 3/6/12 month periods?”
* “What is the company’s tech stack? What existing services you might leverage to simplify the design”

**Example**

“Design a news feed system”

**Candidate:** Jel to za mobile app, web app ili oba?  
**Interviewer:** Oba  
**Candidate:** Koji su neki najbitniji featurei za taj product?  
**Interviewer:** Mogucnost da se postuju i vide friend’s news feed  
**Candidate:** Kako su vesti sortirane? Hronoloski ili na neki drugi nacin? Recimo redosled u kom bi postovi imali razlicitu tezinu, close friends su bitniji od postova neke grupe.   
**Interviewer:** Zarad jednostavnosti, hajde da kazemo da je reverse chronological  
**Candidate:** Koliko prijatelja moze user imati?   
**I:** 5000  
**C:** Koji je traffic volume?  
**I:** 10 million daily active users(DAU)  
**C:** Sta feed sve moze da sadrzi? Media file, media files(video&text), samo text ili sve?

#### Step 2: Propose high level design and get buy-in

U ovom koraku ciljamo na high level design koji ce intervjuer prihvatiti(reach an agreement with the interviewer). Dobra je ideja saradjivati sa intervjuerom.

* Osimisli neki incijalni blueprint, pitaj za feedback. Tretiraj intervjuera kao kolegu i radite zajdno. Vecina dobrih intervjuera vole da pricaju i da se ukljucuju.
* Draw box diagrams with key components na tabli/papiru. To ukljucuje clients, api, web servers, data stores, cache, cdn, message queue etc...
* Odradi back of the envelope calculations da bi evaluirao blueprint koji odgovara scaleu constraintova. Think out loud. Komuniciraj ako je uopste neophodan back of the envelope calculation

Ako je moguce, prodji kroz nekoliko konkretnih use-cases. To ce ti pomoci da malo konkretizujes high level design ili da nadjes neke edge cases koje nisi razmotrio.

Should we include API endpoints & db schema here? To zavisi od problema. Za velike design probleme poput “design google search engine”, to je malo previse low level. Za problem poput dizajniranja backenda za multi-player poker game, to je skroz fer.

**Example:**

Let us use “Design a news feed system” to demonstrate how to approach the high level design.

Na high levelu, design je razdeljen u x2 flow-a=> **feed publishing & news feed building**

* **Feed publishing**: kada user publishuje post, corresponding data is written into cache/db & post ce biti populated into friends news feed
* **Newsfeed building**: the news feed is built by aggregating friend’s posts in reverse chronological order

**High level design za feed publishing:**

A diagram of a web service

AI-generated content may be incorrect.

**High level design za feed building**

**A diagram of a web browser

AI-generated content may be incorrect.**

#### Step 3 - Design deep dive

Ako dodjes do ovog koraka, to znaci da ste se dogovorili na sledecem:

* overall goals & feature scope
* sketch high level blueprinta
* dobio si dovoljno feedbacka na high level design
* imao si neke inicijalne ideje o areas to focus on in deep dive based on her feedback

Sada ces raditi zajedno sa intervjuerom da bi identifikovao i prioritizovao komponente u arhitekturi. Nekada ce interviewer dati hints da voli fokusiranje na high level design. Nekada za senior candidate interview, discussion could be on the system performance characteristics(bottlenecks & resource estimations). U vecini slucajeva, interviewer ze verovatno zeleti da udjes u detalje neke od tvojih komponenti.

Za URL shortener, interesantnoje je uci u hash funkction design koji konvertuje dugacki URL u short URL. Za chat sistem, kako bi se smanjila latencija i podrska online/offline statusa.

Time management je esencijalan pa je lako nekada otici sa najmanjim detaljima koji ne demonstriraju tvoje vestine. Pokusaj da ne ulazis u nepotrebne detalje, npr ulazak u EdgeRank algoritam za Facebook feed ranking. Tema je dizajniranje sistema, a ne algoritma.

**Example step-a; napokon:**

Sada cemo investigate dva najbitnija use cases: (bice u chapter 11 objasnjeni)

* feed publishing
* news feed retrieval

**Feed publishing**

A diagram of a web service

AI-generated content may be incorrect.

**News feed retrieval:**

**A diagram of a software development

AI-generated content may be incorrect.**

#### Step 4 - wrap up

Ovde te mozda pita par follow up questions ili ti dao slobodu da razgovaras o dodatnim tackama. Par smernica:

* Intervjuer mozda zeli da identifikujes bottlenecks & discuss potential improvements. Nikad ne reci da ti je dizajn savrsen :)
* Moze biti korisno da das recap tvog dizajna. Ovo je itekako bitno ako si predlozio nekoliko resenja. Refreshing your interviewer’s memory can be helpful after a long session.
* Error cases(server failure, network loss) su interesantni
* Operacioni problemi, kako monitoringujes metrice i error logs? Kako odraditi roll out
* Kako handleovati sledeci scale curve je takodje interesantno. Npr ako tvoj trenutni dizjn podrzava 1 milion korisnika, koje izmene moras napraviti za bi ti podrzavao 10 miliona korisnika.
* Predlozi druge refinemente

**Radi:**

* Uvek pitaj za razjasnjenje. Nemoj racunati da su tvoje pretpostavke tacne.
* Razume zahteve problema
* Nema tacnog/netacnog/najboljeg resenja. Resenje dizajnirano da se resi problem malog startupa je drugacije od onog sto ima milione korisnika.
* Dozvoli da intervjuer zna o cemu razmisljas
* Predlozi nekoliko pristupa ako je to moguce
* Jednom kada se dogovorite an blueprint, otidi u detalje na svaku komponentu. Dizajniraj critical components first.
* Bounce ideas off the interviewer. Dobar intervjuer ce raditi sa tobom kao teammate
* Nikad ne odustaj :)
* Uvek pitaj za feedback. nemoj da mislis da ti je intervju zavrsen samo zato sto si dao neki dizajn. Nisi gotov dok intervjuer ne kaze da ste gotovi. I am the one that knocks.

**Ne radi:**

* Ne budi nespreman za tipicna intervju pitanja
* Ne skaci na resenja bez da se razjasne zahtevi i pretpostavke
* Ne idi u previse detalja na jednu komponentu. Dozvoli high level design first, a onda drill down
* Ako se zaglavis, nemoj se stideti i pitaj za hints
* Komuniciraj, ne razmisljas “u sebi”

**Time allocation:**

Cesto dobijes 45 min ili 1h da ih odradis sto nije dovoljno da se pokrije ceo dizajn. Time management je bitan.

Step 1 - razumi problem i dobij design scope; 3-10 min  
Step 2 - predlozi high level design & get buy-in; 10-15 min  
Step 3 - dizajniraj deep dive; 10-25 min  
Step 4 - wrap up; 3-5 min

## Chapter 4: Design a rate limiter

In a network system, rate limiter se koristi da se kontrolise “rate of traffic” sent by a client or a service. Znaci koristi se da bi se kontrolisao saobracaj koji salje client ili servis.

U HTTP svetu, rate limiter ogranicava broj client requestova koji smeju da se posalju kroz odredjeni period.

Ako API Request count prevazidje odredjeni granicnik(threshold) definisan od strane rate limitera, svi naredni pozivi su blokirani.  
Par primera:

* User ne moze da pise vise od 2 posta po sekundi
* Mozes napraviti maximum 10 accountova dnevno od strane iste IP adrese
* Mozes claimovati ne vise od 5 nagrada nedeljno sa istog uredjaja

Koji su benefiti koriscenja rate limitera:

* Prevencija DoS napada(namernih/nenamernih) blokiranjem viska poziva. Skoro svi moderni sistemi imaju neki vid rate limitera, npr twitter ogranicava broj tweetova na 300/3h. Google docs API imaju limit od 300/user/60sec for read requests.
* Smanjuje cenu na taj nacin sto ce ti trebati sad manje servera za alociju resursa high priority API-evima. Takodje je jako bitno za kompanije koje koriste npr 3rd party API i ako te neko naplacuje na per call basis.
* Prevencija servera da ne budu overloaded, rate limiter se koristi da filtrira visak zahteva koje mogu napraviti nemarni korisnici ili botovi.

#### Step 1 - Understand the problem and establish design scope

Mogu se koristiti razni algoritmi za implementaciju rate limitera svaki sa svojim prednostima i manama. Interakcijom izmedju tebe i intervjuera mozes doci do razresenja.

**C:** Koji tip rate limitera cemo dizajnirati. Jel u pitanju client-side ili server-side API rate limiter?  
**I:** Fokusiracemo se na server-side API rate limiter  
C: Na osnovu cega ce limitovati requestove(throttle) - IP, user ID ili druga svojstva?  
I: Treba da bude dovoljno fleksibilan da podrzava razlicite skupove pravila  
C: Koji je scale sistema, da li je u pitanju startup ili velika firma sa velikom bazom usera  
I: Sistem mora da podrzi veliki broj zahteva  
C: Da li ce sistem biti u distribuiranom okruzenju  
I: Da  
C: Da li ce rate limiter biti zaseban servis ili ce se implementirati unutar aplikacije  
I: Na tebi je  
C: Da li moramo da obavestimo korisnike koji su throttleovani  
I: Da

Requirements:

* Precizno ogranici visak zahteva
* Niska latencija. Ne bi trebao da uspori HTTP response time
* Koristi sto manje memorije
* Distribuirani rate limiting. Rate limiter bi trebao da moze biti deljen izmedju vise servera ili procesa
* Exception handling. Pokazi exceptione korisnicima koji su throttled
* High fault tolerance. Ako ima problema sa limiterom(npr cache server ode offline), to nece zahvatiti ceo sistem.

#### Step 2 - high level design & get buy in

Hajde simple primer sa basic client->server modelom za komunikaciju

**Gde postaviti rate limiter?**

Intuitivno, moze na client, a moze i na server-side.

* **Client side implementacija.** Client je generalno nepouzdan za enforceovanje rate limitinga jer client requests mogu lako da se modifikuju od strane malicioznih korisnika. Takodje mozda nemamo ni kontrolu nad client implementacijom.
* **Server side implementacija.** Desava se na backendu(samom api server), ispod je slika primera  
  **A close-up of a message

  AI-generated content may be incorrect.**

**Tu takodje postoji i alternativa.** Mozemo napraviti **rate limiter middleware** koji throttleuje requestove ka tvojim APIevima.

A line with a line pointing to a line

AI-generated content may be incorrect.

Kako bi to radilo? Recimo da API dozvoljava 2 req/sec i client salje 3 req ka serveru u sekundi. Prva dva su routeovana u API server, ali na trecem rate limiter middleware throttleuje i **vraca HTTP status code 429(too many requests)**

**A diagram of a line with arrows and a red x

AI-generated content may be incorrect.**

Cloud microservices kako su postali jako popularni, rate limiteri se tu cesto implementiraju unutar komponente “API Gateway”. API Gateway je fully managed service koja podrzava rate limiting, SSL termination, authentication, IP whitelisting, servicing static content, etc.

Za sada je dovoljno znati da je API Gateway middleware koji podrzava rate limiting.

Pri dizajniranju rate limitera, bitno pitanje koje sebi moramo postaviti: **Gde treba rate limiter da bude implementiran? Server side ili gateway?**

Ne postoji “absolute answer”. Zavisi od trenutnog stacka firme, resursa, prioriteta, ciljeva i slicno. Evo sta mozes da uradis kako bi sebi pomogao:

* Evaluiraj trenutni tech stack - programski jezici, cache servisi i slicno. Garantuj da ti je trenutni programming language efikasan za implementaciju rate limitinga na serveru.
* Identifikuj rate limiting algoritam koji ti ispunjava pravila poslovanja(business needs). Kada implementiras sve na serveru, imas punu kontrolu algoritma. Ukoliko odlucis na 3rd party gateway, bices ogranicen.
* Ako si koristio microservice arhitekturu i ako je imala api gateway u dizajnu zarad autentikacije/ip whitelisting i slicno, mozda budes morao dodati tu rate limiter.
* Pravljenje svog rate limiting servisa zahteva vreme. Ako nemas dovoljno engineering resursa za implementaciju limitera, komercijalni api gateway je bolja opcija.

**Algoritmi za rate limiting**

* **token bucket**
* **leaking bucket**
* **fixed window counter**
* **sliding window log**
* **sliding window counter**

##### Token bucket algoritam:

Cesto se koristi, jednostavan je i koristi se i u amazonu i stripe-u kao algoritam za throttle API requestova.

Token bucket je container preferinisanog kapaciteta. Tokeni se postavljaju u bucket**(kantu)** periodicno po unapred odredjenim rateovima. Jednom kad je bucket pun, ne dodavaju se vise tokeni.

U slici ispod, token bucket capacity je 4. Refiller stavlja 2 tokena u bucket svake sekunde. Jednom kad je bucket full, visak tokena ce overflowovati.

A diagram of a refilling process

AI-generated content may be incorrect.

Svaki request konzumira x1 token. Kada request stigne, proveravamo da li ima dovljno tokena u bucketu.

* Ako ima dovoljno tokena, izvadicemo po x1 token za svaki request i request prolazi
* Ako nema dovoljno dokena, request je droppovan

A diagram of a blockchain process

AI-generated content may be incorrect.

U narednoj slici(4-6) prikazan je refill & rate limiting logic. Bucket size je 4, a refill rate je 4 tokena po minutu.

A screenshot of a computer screen

AI-generated content may be incorrect.

Token bucket algoritam uzima x2 parametra:

* **Bucket size:** Maximum number of tokens dozvoljen u bucketu
* **Refill rate:** Broj tokena koji se postavljaju u bucket svake sekunde

Koliko nam bucketa treba? To zavisi od rate limiting rules.

* **Cesto je neophodno da imas razlicite buckete za razlicite API endpointe.** Npr ako korisnik sme da pravi x1 post/second, doda 150 prijatelja na dan i likeuje 5 postova po sekundi, x3 bucketa su neophodna po korisniku.
* **Ako treba da throttleujemo requestove na osnovu IP adresa, svaka IP adresa ce zahtevati Bucket**.
* **Ako sistem dozvoljava max 10k req/sec**, ima smisla imati global bucket koji dele svi requestovi.

**Prednosti**:  
- Jednostavan algoritam  
- Memorijski efikasno  
- Token bucket dozvoljava burst of traffic za kratke periode. Request moze proci dokle god postoje tokeni

**Mane:**  
- x2 parametra u alrogirmu su bucket size & token refill rate koji mogu biti izazovni za ispravno podesavanje(tuning them properly)

##### Leaking bucket algoritam:

Slican je tocken bucketu osim sto se requestovi procesiraju u “fixed rate”. To znci da se tu koristi FIFO queue.

Kako radi:   
- Kad stigne request, system proverava da li je queue pun. Ako nije, request ce biti dodat.   
- Ako je queue pun, request se droppuje.   
- Requestovi se povlace iz queue I procesuiraju u regularnim intervalima

A diagram of a flowchart

AI-generated content may be incorrect.

Prihvata naredna dva parametra:  
- **Bucket size**: ovo je u sustini queue size koji drzi requestove sto treba procesovati  
- **Outflow rate**: Definise Koliko requestova se moze procesirati na fixed rate(cesto /s)

Shopify koristi leaking bucket algoritam za rate limiting.

Prednosti:  
- memorijski efikasno  
- requestovi su procesirani na fixed rate pak je dobro za slucajeve kada ti je neophodan stabilan outflow rate.

Mane:  
- Ne mozes raditi burst requestova  
- Postoje x2 parametra pa je tuning problem

##### Fixed window counter algoritam

Ovako radi:

* Algoritam deli timeline u fix-size time windows I dodeljuje brojac za svaki window.
* Svaki request inkrementira brojac za +1
* Jednom kada counter dosegne pre-defined threshold, novi requestovi su dropped dok ne zapocne novi window.

A graph showing the results of a successful success

AI-generated content may be incorrect.

Veliki problem sa ovim algoritmom jeste da burst of traffic koji moze da se desi na samim ivicama ovih time window ace da izazove prolazenje od vise requestova nego sto je planirano.

A diagram of a diagram with text

AI-generated content may be incorrect.

Ako ide x10 requestova u 2:00:59 pa x10 requestova u novom time window 2:01:01, to je x20 requestova u 3 sekunde umesto x10 requestova po minuti.

Prednosti:

* Lak za razumevanje
* Memorijski efikasno
* Resetting available quota at the end of a unit time window fits certain use cases

Mane:

* Spike u saobracaju na ivicama window-a moze rezultirati u vise problema

##### Sliding window log algoritam

Problem prethodnog algoritma je upravo taj ako dodjes do ivice time window-a. Sliding window log algoritam to popravlja.

Kako radi:

* Algoritam prati “request timestamps”(algoritam pecatira zahteve). Timestamp data se cesto cuva u cache-u(sorted set Redis-a).
* Kada stigne novi request, ukloni sve outdated timestamps. Outdatet timestamps su definisani kao oni sto su stariji od pocetka trenutnog time window-a.
* Dodaj timestamp za novi request u log.
* Ako je velicina loga ista ili manja od dozvoljenog counta, request se acceptuje. U suprotnom reject.

Ok ok ok

Ono sto se radi. Posalje se zahtev. Gleda se vreme kada je zahtev poslat. Imas timeframe I limit; timeframe je 1 min, limit je 2 requests.

Kad ti request stigne u 1:01:40, gleda se sta je pristizalo u zadnjih minut. Znaci ako od 1:00:40 – 1:01:40 nemas vise od 2 requesta, onda se acceptuje. Stize request u 41, isto tako se gleda je lima ili nema u zadnjih minut “limit” ispunjen.

Svi requestovi koji su pre 1:00:40, oni su u momentu 1:01:40 outdated requestovi I ne racunaju se. Ovo “vreme” koje spominjem se u redisu cuva u unix timestampu cesto. Samo predstavljam format lak za citanje.

Znaci ovaj algoritam je “dinamican” jer racuna timeframe u zavisnosti toga kada je stigao zahtev.

Prednosti:

* Rate limiting je u ovom slucaju jako precizan. U bilo kom timelineu, request nece navrsiti time/request limit

Mane:

* Konzumira puno memorije. Cak iako je request odbijen, njegov timestamp se mora storeovati u memoriji.

##### Sliding window counter algorithm

Ovo je hibridni pristup koji kombinuje fixed windows counter pristup I sliding window log.

Moze se implamentirati na dva nacina. Ovde cemo samo jedan implementirati jer jbg, moze nam se.

A diagram of a diagram of a rolling minute

AI-generated content may be incorrect.

Recimo da rate limiter treba da dozvoli max 7 requestova po minuti I tu je 5 requestova u prethodnoj minuti, a 3 requesta u trenutnoj minuti. Za novi request koji stize na 30% poziciji trenutnog minuta, broj requestova u rolling window se racuna koriscenjem sledece formule:

* Requests in current window + requests in the previous window \* overlap percentage of the rolling windows & previous window

Koriscenjem formule: 3+5\*0.7% = 6.5requests.

U zavisnosti od slucaja, broj moze biti zaokrugljen gore ili dole. Mi cemo ga floor na 6.

S obzirom da je 7 max, a mi smo na 6, dozvolicemo taj request.

Meni se licno ovaj algoritam cini komplikovanim(Petrovo misljenje)

Prednosti:

* Pegla skokove saobracaja jer se rate bazira na proseku prethodnog prozora.
* Memorijski je efikasan

Mane:

* Radi samo za “ne toliko striktne” slucajeve. Aproksimacija je u sustini problem jer prozori nisu podjednako rasporedjeni, ali u eksperimentu Cloudflare-a, ispostavilo se da samo 0.003% requestova je pusteno, a nije trebalo.

##### High level architecture

Znaci rate limiter broji zahteve I naspram odredjenih granicnika ih propusta ili odbija. Trebamo negde sad da cuvamo ta prebrojavanja(ip address/userid requests) I za to se cesto koristi REDIS cache jer je baza previse spora.   
Ovo su dve REDIS komande koje ce nam trebati:

* **INCR** command za povecavanje +1
* **EXPIRE** podesava expiry nekog podatka(countera). Ako je expired, counter se automatski brise

A diagram of a diagram

AI-generated content may be incorrect.

Client ce da salje zahteve rate limiter middlewareu, middleware fetchuje counter I proverava stanje limitera. Ako je limit reached, odbija request. Ako nije, inkrementira, upisuje novu vrednost I request se salje na API server.

#### Step 3 – Design deep dive

High level diagram ne odgovara na pitanja:  
- kako se rate limiting pravila kreiraju i gde se pravila cuvaju  
- kako handleovati requestove koji su rate limited

##### Rate limiting rules

Lyft je open-sourceovao svoju rate limiting komponentu pa to mozemo koristiti kao referencu.

A close-up of a computer code

AI-generated content may be incorrect.

Ovde je rate limiter podesen za da marketing prima max 5 poruka na dan.

A screenshot of a computer program

AI-generated content may be incorrect.

U ovom pravilu client npr ne sme da se uloguje vise od 5 puta u minuti

##### Exceeding rate limit

Kada se prevazidje rate limit, API vraca HTTP response code 429(too many requests). Nekada u zavisnosti od situacije, rate limited requestova mozda cak i zelimo da sacuvamo za kasnije jer npr, ako su neke porudzbine rate limitovane zbog system overloada, mozda te porudzbine zelimo da prihvatimo kasnije.

##### Rate limiter headers

Sa njima mozemo da kazemo clientu broj dozvoljenih/preostalih requestova ili da je client throttleovan.

***X-Ratelimit-Remaining*** – koliko je dozvoljenih poziva preostalo

***X-Ratelimit-Limit*** – koliko poziva uopsteno smes po “time window”-u

***X-Ratelimit-Retry-After*** – Koilko sekundi moras cekati dok si untrhottled

***X-Ratelimit-Retry-After*** se u sustini vraca samo kada je u pitanju 429 response.

##### Detailed design

A diagram of a software flow

AI-generated content may be incorrect.

Rate limiting rules se cuvaju negde na disku. Workeri povremeno pulluju rules sa diska i storeuju ih u cache. Kada client posalje request severu, request prvo pristize na rate limiter.

Rate limiter ucitava pravila iz cache-a, fetchuje counter i poslednji request timestamp iz redis cache-a i odredjuje da li propusta ili ne prepopusta.

##### Rate limiter u distribuiranom okruzenju

Pravljenje rate limitera koji radi za single server environment je lagano. Problemi **race conditiona** i **sinhronizacije** su problemi sa kojima se susreces u distribuiranim sistemima.

Isto kao i react state, mi imamo npr neki request, on stigne na rate limiter i zahteva update countera. Ako x2 requesta updateuju counter u “istu vrednost” npr “4”, mi smo izgubili jedan upis.

A diagram of a computer

AI-generated content may be incorrect.

Ako x2 requesta konkurentno citaju counter value pre neko jedan upise, to od gore ce se desiti.

**Lock** je nego logicno resenje, ali znaj da ce to usporiti sistem.   
Postoje x2 strategije za resavanje ovog problema: (necemo o tome ovde)  
- Lua Script  
- Sorted sets data structure in Redis

**Synchronization issue**

Kako bi se podrzavali milioni korisnika, jedan rate limiter sever mozda nece biti dovoljan, a kada se koristi vise rate limiter servera, sinhronizacija je “must”.

Web tier je stateless. Client bi trebalo da moze da salje requestove razlicitim rate limiterima ka na sledecoj slici

A diagram of a customer service

AI-generated content may be incorrect.

Ako se sinhronizacija ne desi, rate limiter 1 nece sadrzati podatke o rate limiteru 2.

Jedno resenje bi bilo npr koriscenje sticky sesija koje dozvoljava clientu da salje saobracaj istom rate limiteru. Cesto se ovo resenje NE PREPORUCUJE. Nije skalabilno, nije fleksibilno.

Resenje bi bilo centralized data stores poput Redisa.

A diagram of a rate limiter

AI-generated content may be incorrect.

##### Performance optimization

Ovo je cesta tema u sys design intervjuima.

Prvo, multi-data center setup je krucijalan za rate limiter jer latencija ce biti problem za daleke korisnike.

Drugo, sinhronizuj podatke sa eventual consistency modelom. (chapter 6, bice)

##### Monitoring

Nakon sto stavis rate limiter, sad je bitno prikupljati podatke da li je efektivan.

Zelimo da znamo da li je rate limiting algoritam efetktivan i da li su pravila efektivna.

Ako su rate limiting pravila npr previse ogranicavajuca, mnogo validnih requestova se droppuje. Tada cemo zeleti smanjiti tj povecati granice. Takodje ako vidimo da je rate limiter neefikasan pri naglom povecanju saobracaja, mozda zelimo da promenimo algoritam u npr tocken bucket koji podrzava veliki burst saobracaja.

#### Step 4 – Wrap up

Ovde smo pricali o razlicitim algoritmima rate limitinga, prednosti i mane.

Potom o arhitekturi softvera, rate limiter u distribuiranom okruzenju, perf optimization & monitoring.

Na kraju, postoji jos nekoliko talking points koje mozes pomenuti ako ti vreme dozvoli:

* Hard vs soft rate limiting
  + Hard: broj zahteva ne sme prekoraciti granice
  + Soft: broj zahteva sme kratkorocno prekoraciti granice
* Rate limiting na razlicitim nivoima
  + Ovde smo pricali samo na HTTP layeru. Moguce je raditi rate limiting i na drugim layerima OSI modela(IP address koriscenjem IP Tables)
* Izbegavaj da budes rate limited. Dizajniraj cliente sa best practices
  + Koristi client cache za izbegavanje api calls
  + Razumi limit & ne salji too many requests u kratkom time frameu
  + Include code to catch exceptions/erros kako bi se clienti recoverovali
  + Dodaj back off time i retry logiku

## Chapter 5: Design Consistent Hashing

Consistent hashing je tehnika koja se koristi za efektino horizontalno skaliranje distribuiranih sistema jer omogucava distribuciju zahteva efikasno i podjednako kroz servere.

### Rehashing problem

Ako imas ***n*** cache servera, cest nacin za balansiranje opterecenja jeste koriscenje sledece hash metode:

***serverIndex = hash(key) % N;*** gde je N velicina server pool-a.   
Scenario: imamo x4 servera i x8 string keys sa njihovim hashovima.

A table with numbers and a few tables

AI-generated content may be incorrect.

Da fetchujes server gde se key nalazi, odradicemo onu gore funkciju, dakle treba nam kljuc 0 i to ce ici ***hash(key0)%4) = 1*** sto znaci da se key nalazi u serveru 1.

A number of numbers and symbols

AI-generated content may be incorrect.

Ova metoda ce super funkcionisati kada je velicina sever pool-a fiksirana i kada se distribucija podataka podjednako rasporedjuje(celebrity problem, seti se).

Problem je takodje ako server 1 ode offline ili ako se doda novi server. Ukoliko oduzmemo jedan server iz racunice, onda dobijamo drugacije rezultate.

A screenshot of a computer

AI-generated content may be incorrect.

Vecina kljuceva se redistribuira. To znaci da kada server 1 ode offline, vecina cache clienata ce se povezati na pogresan server za fetch podataka sto izaziva mnogo cache misses. Consistent hashing je efektivna tehnika za otklanjanje ovog problema.

### Consistent Hashing

Wikipedia: Consistent Hashing je tip hashinga takvog da kada je hash tabela resized i consistent hashing se koristi, samo k/n keys moraju biti remapirani u proseku; k je broj kljuceva, a n je broj slotova.

In contrast, u vecini tradicionalnih hash tabela, promena u broju array slotova cesto zahteva da se svi kljucevi remappiraju.

### Hash space & Hash rings

Kako radi consistent hashing. Recimo da se SHA-1 hashing funkcija koristi kao ***f*** i output hash funkcije je: x0, x1, x2, … xn.

U kriptografiji SHA-1 hash space ide od 0 do 2^160.   
To znaci da x0(x0 odgovara broju 0), a xN je 2^160 – 1. Sve ostale vrednosti su negde u tom krugu(hash space-u).

x0 xN  
  
I onda dobijemo prsten.

A diagram of a circle with arrows pointing at the center

AI-generated content may be incorrect.

### Hash servers

Koriscenjem iste hash funkcije, mapiracemo servere na osnovu IP-a ili imena u tom prstenu. Nasa x4 servera ce izgledati nekako ovako.

A diagram of a diagram

AI-generated content may be incorrect.

### Hash keys

Funkcija koja je koristena ovde je drugacija od one hash funkcije koju si gore video. Ono gore je “moduo” pa se na osnovu toga racuna broj servera. Ovde nema modular(moduo) operatora. Ovde ce se i hash keyevi hashovati na prsten.

A circular diagram with colored circles and numbers

AI-generated content may be incorrect.

### Server lookup

Da se pronadje koji key se cuva na kom serveru, idemo smerom kazaljke na satu od key pozicije pa dok se ne pronadje server. A diagram of a circle with arrows and points

AI-generated content may be incorrect.

### Add a server

Dodavanje novog servera ce zahtevati redistribuciju samo jednog dela kljuceva. U sustini to su svi kljucevi koji su u hash circle pre njega, a posle prethodnog servera. A diagram of a circle with numbers and points

AI-generated content may be incorrect.

### Remove a server

Ista prica kao gore. Samo kljuceva ce se re-routeovati dok ostali kljucevi nece biti takntui.

A diagram of a circle with numbers and points

AI-generated content may be incorrect.

### Two issues in the basic approach

Ovaj pristup consistent hashingu ima x2 problema.

1. Nemoguce je zadrzati istu velicinu particija na prstenu s obzirom da se serveri mogu dodavati i oduzimati. Jedna particija moze biti 5 puta veca od druge.
2. Moguce je imati neuniformisanu distribuciju kljuceva u prstenu. Da se vecina kljuceva nalazi na x1 serveru, a ostali da su prazni.

Tehnikom “virtuelni node” se to sve resava.

### Virtuelni Nodes

Virtuelni node ima referencu na pravi node. Pravi node je u sustini jedan “server” u nasem distribuiranom okruzenju.

Virtuelnih node-ova imamo vise i npr sad da imamo x2 prava servera s0 i s1, mi cemo imati za svaki po x3 virtuelna nodea i to ce biti:  
s0\_0; s0\_1; s0\_2;  
s1\_0; s1\_1; s1\_2;

U nasem krugu ici ce ovako segmentacije; (s0\_0, s1\_0), (s0\_1, s1\_1), (s0\_2, s1\_2)

A diagram of a circle with circles and lines

AI-generated content may be incorrect.

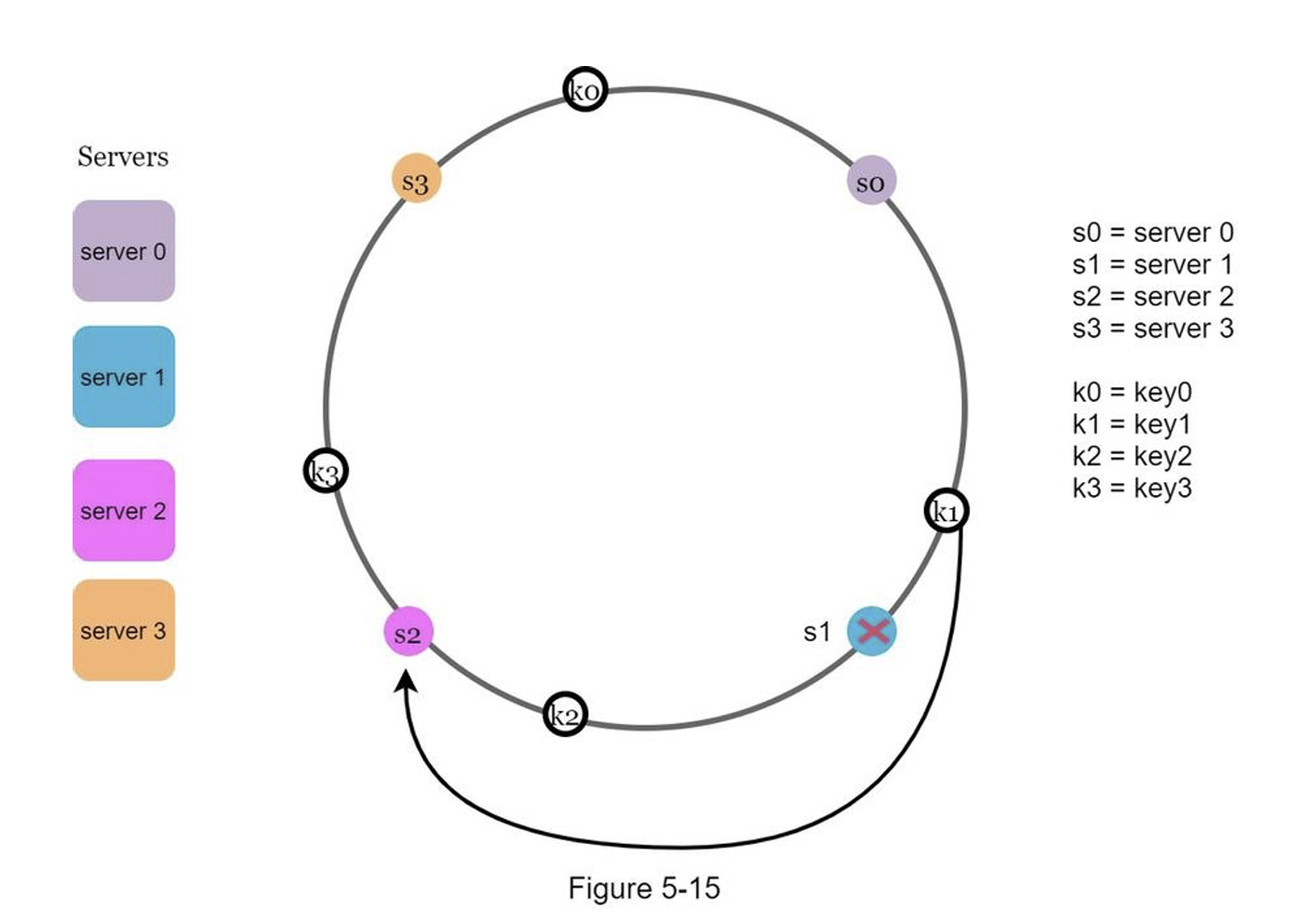
Logika postavljanja kljuceva i slicno je i dalje ista. Dakle, imamo kljuc k0 i on ce se cuvati u node-u prvi na koji naleti u smeru desno kazaljke na satu.

Sto je vise virtuelnih node-ova, to je bolja distribucija kljuceva. Devijacija u kljucevima u nekom istrazivanju je na 5-10% u zavisnosti od broja virtuelnih node-ova. Koristilo se oko 100-200 nodeova i sto ih imas vise, to ce devijacija biti manja. Jedini problem – trebas skladistiti mapiranja.

### Find affected keys

Znaci, kada se desi da server padne ili da se doda mi cemo deo kljuceva morati redistribuirati. Ovde cemo pricati o tome kako te kljuceve mozemo pronaci.

U sustini, pronaci ce se tako sto se ide od servera koji je otklonjen/dodat i radices iteraciju kontra smeru kazaljke na satu sve do prvog servera. U prevodu, brojis “u nazad” i sve kljuceve znas da sad moras prebaciti u drugi server(being the next server or the new server if new is addedd).



Ovde se server s1 ugasio i sada svi kljucevi koji se nalaze u kontra smeru kazaljke na satu od s2 ce biti dodati u s2, a da su posle servera s0.

Wrap up:

* Ovim se otklanja dosta problema, cak i kod shardinga gde si imao celebrity problem, na ovaj nacin se to otklanja jer se podaci bolje rasporedjuju.
* Lako horizontalno skaliranje

## Chapter 6: Design a key value store

Key-value store je u sustini key-value database.   
Kljuc treba da je unique i moze da se cuva kao string ili hash, a value je objekat bilo koje vrste(int/array/obj i slicno). Paritet kljuca i vrednosti se naziva key-value pair.

U ovom zadatku key value store treba da podrzava x2 operacije:  
- put(key, object)  
- get(key) :object

### Step 1: Understand the problem and establish design scope

U ovom key-value store mi zelimo da mozemo:

* Store big data.
* High availability. Reponds quickly even with failures.
* High scalability. Can be scaled to support high data sets.
* Size of key-val pair is small(10KB).
* Tunable consistency.
* Low latency.

#### Single server key-value store

Konfiguracija kojom bi se ovo resilo kroz single server u sustini ne postoji jer single server podrazumeva da bi se kljucevi verovatno cuvali u nekoj hash tabeli(data structure) i tako bi im se i pristupalo. Iako je pristup relativno brz, setup lak, problem koji ovde nastaje jeste sto se taj prostor jako lako moze popuniti i to jako brzo. To mnogo zavisi od ram memorije.

Takodje ono sto mozes da uradis kao deo optimizacije jeste da vrsis kompresije podataka i da samo cesto pristupane podatke cuvas u memoriji, a ostale na disku.

U svakom slucaju, nama ovde treba distribuirani key-value store.

#### Distributed key-value store

Drugi naziv za distributed key-value store je: **distributed hash table**